Dynamic programming

Code: https://github.com/mesmere/sasha-tutorial/tree/main/tba-1

What is dynamic programming?

When the problem:

1. Has a **recursive structure**…
2. …with **overlapping subproblems**,

Approach it by:

1. Writing **a recurrence**…
2. …and writing code to **populate a table**!

Application - Making change

Problem:

A vending machine needs to dispense change for some cash amount *x* which has been overpaid by the customer. To minimize cash costs, the machine should dispense **as few coins as possible**. Also, since this design might be deployed to many regions, the algorithm should work with **any denominations of coins**.

United States:	*1c, 5c, 10c, 25c*
European Union:	*1c, 2c, 5c, 10c, 20c, 50c, €1, €2*
Japan:	*¥1, ¥5, ¥10, ¥50, ¥100, ¥500*

Approach:

1. Write a recurrence
2. Write code to populate a table.

Application - Making change

An algorithm which *won't work* is the greedy algorithm; i.e. repeatedly pick up the largest coin that doesn't cause you to overshoot x.

That approach doesn't work in general:

> Imagine a currency where coins come in denominations of 1, 4, 6.
> Try to make change for x=8.
> The greedy algorithm gives us [6, 1, 1], but it's better to dispense [4, 4].

Application - Making change

Let `min_coins(n)` be a function which value is the **minimum number of coins** it takes us to dispense n, given the coin denominations $[c_1, c_2, c_3, \ldots]$. Then:

$$\texttt{min\_coins}(n) = \min \begin{cases} \text{min\_coins}(n - c_1) + 1 & n \geq c_1 \\ \text{min\_coins}(n - c_2) + 1 & n \geq c_2 \\ \text{min\_coins}(n - c_3) + 1 & n \geq c_3 \\ \ldots & \\ 0 & \mathrm{n} = 0 \end{cases} .$$

This is our **recurrence**.

Application - Making change

Let's make the problem concrete by plugging in U.S. currency [1, 5, 10, 25] for our coin denominations.

$$\texttt{min\_coins}(n) = \min \begin{cases} \text{min\_coins}(n-1)+1 & n \geq 1 \\ \text{min\_coins}(n-5)+1 & n \geq 5 \\ \text{min\_coins}(n-10)+1 & n \geq 10 \\ \text{min\_coins}(n-25)+1 & n \geq 25 \\ 0 & \text{n} = 0 \end{cases} .$$

The next step is to populate a table containing values of `min_coins(`n`)`, starting at n=0.

Application - Making change

$$\texttt{min\_coins}(n) = \min \begin{cases} \text{min\_coins}(n - 1) + 1 & n \geq 1 \\ \text{min\_coins}(n - 5) + 1 & n \geq 5 \\ \text{min\_coins}(n - 10) + 1 & n \geq 10 \\ \text{min\_coins}(n - 25) + 1 & n \geq 25 \\ 0 & n = 0 \end{cases}.$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`min_coins(n)`	0	1	2	3	4	1	2	3	4	5	1	2	3	4	5	2	3	4
`next coin`		1	1	1	1	5	1	1	1	1	10	1	1	1	1	5	1	1

Populate the table up to the desired *n*. `next coin` tracks which branch of the “min” we took at each step of *n*, so that we can rewind back through the table to reconstruct the actual change to dispense.

Application - Subset sum

Problem:

Given a multiset of weighted items $S = \{ w_1, w_2, \dots w_n \}$ and an upper bound B, find a subset $S' \subseteq S$ such that $\Sigma S'$ is maximized without exceeding B.

Approach:

1. Write a recurrence.
2. Write code to populate a table.

Application - Subset sum

Let **L**(i, b) be the largest sum we can obtain using a subset of $\{ w_1, w_2, \ldots w_i \}$, with upper bound b.

$$
\begin{aligned}
\mathbf{L}(0, b) &= 0 \\
\mathbf{L}(i, b) &= \mathbf{L}(i-1, b) && \text{if } 1 \le i \le n \text{ and } b < w_i \\
\mathbf{L}(i, b) &= \max(\mathbf{L}(i-1, b-w_i)+w_i, \mathbf{L}(i-1, b)) && \text{if } 1 \le i \le n \text{ and } b \ge w_i
\end{aligned}
$$

The blue case says that we're using w_i in the packing. So we take the best we could do without w_i and without the space we'll need for w_i, and to it we add w_i.

The red case says that we're not using w_i in the packing. So we just take the best we could do without w_i, with unchanged free space.

Application - Subset sum

```
for b from 0 to B:
    table[0][b] = 0

for i from 1 to n:
    for b from 1 to B:
        if b<w_i:
            table[i][b] = table[i-1][b]
        else:
            table[i][b] = max(table[i-1][b-w_i]+w_i, table[i-1][b])
```

Now we're interested in `table[n][B]`. This is the largest sum we can obtain with all *n* items available to us, while staying under the full B upper bound.

But what's the actual packing? All we have so far is a number... 😔

`subsetSum([5,8,9,10,11,12,13,14],20);`

	b=0	b=1	b=2	b=3	b=4	b=5	b=6	b=7	b=8	b=9	b=10	b=11	b=12	b=13	b=14	b=15	b=16	b=17	b=18	b=19	b=20
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1 (5)	0	0	0	0	0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
i=2 (8)	0	0	0	0	0	5	5	5	8	8	8	8	8	13	13	13	13	13	13	13	13
i=3 (**9**)	0	0	0	0	0	5	5	5	8	9	9	9	9	13	14	14	14	17	17	17	17
i=4 (10)	0	0	0	0	0	5	5	5	8	9	10	10	10	13	14	15	16	17	18	19	19
i=5 (**11**)	0	0	0	0	0	5	5	5	8	9	10	11	11	13	14	15	16	17	18	19	20
i=6 (12)	0	0	0	0	0	5	5	5	8	9	10	11	12	13	14	15	16	17	18	19	20
i=7 (13)	0	0	0	0	0	5	5	5	8	9	10	11	12	13	14	15	16	17	18	19	20
i=8 (14)	0	0	0	0	0	5	5	5	8	9	10	11	12	13	14	15	16	17	18	19	20

$$L(i, b) = \max(L(i-1, b-w_i)+w_i, L(i-1, b))$$

Application - Subset sum

Question: What's the running time of this thing?

1. $\Theta(n \cdot B)$ to build the table.
2. $\Theta(n)$ to trace through the table and reconstruct the packing.

But… you've been bamboozled. This is actually *technically* still exponential time.

Application - All-pairs shortest path

Problem:

Given a weighted graph, find **the lengths of the shortest paths** between **every pair of vertices**.

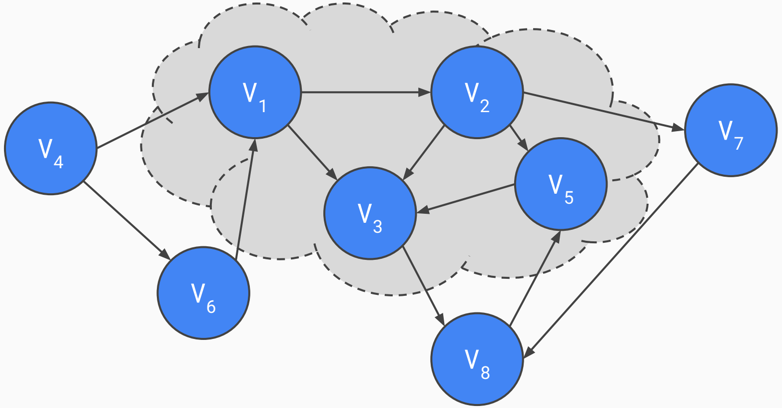
Approach:

1. Write a recurrence.
2. Write code to populate a table.

Application - All-pairs shortest path

Floyd's algorithm approach:

Let $D_{ij}^{(k)}$ be the shortest possible path distance from V_i to V_j using only $\{V_1, V_2, \dots V_k\}$ as "internal vertices."



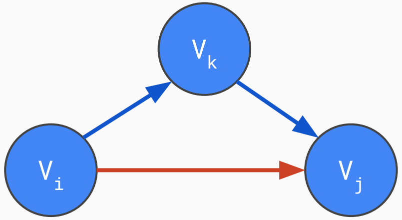
Example showing $D_{4,7}^{(5)}$. We want the length of the shortest path from V_4 to V_7 using only the first 5 vertices.

Application - All-pairs shortest path

$$D_{ij}^{(0)} = w_{ij} \qquad \text{if } (i, j) \ni E$$

$$D_{ij}^{(0)} = \infty \qquad \text{if } (i, j) \not\ni E$$

$$D_{ij}^{(k)} = \min(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}) \qquad \text{if } k>0$$



Each time we move on to allowing the k^{th} vertex, we need to decide whether to include V_k in our path from V_i to V_j.
Take the best known paths $V_i \rightarrow V_k$ and then $V_k \rightarrow V_j$, or keep our old best path $V_i \rightarrow V_k$?

Application - All-pairs shortest path

```
for i from 1 to |V|:
    for j from 1 to |V|:
        table[i][j][0] = w_{i,j}
for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            table[i][j][k] = min(table[i][j][k-1],
                      table[i][k][k-1] + table[k][j][k-1])
```

Application - All-pairs shortest path

```
// Cute syntax for indicating no edge.
const X = Number.POSITIVE_INFINITY;

// |V| by |V| array with each entry i,j holding the weight of the edge from V_i to V_j.
const table = [
  [ 0, X, X, 1, 5 ],
  [ 1, 0, 9, X, X ],
  [ 3, 1, 0, X, X ],
  [ X, X, X, 0, 3 ],
  [ X, X, 2, X, 0 ],
];

// Number of vertices in the graph.
const n = table.length;

// Repeatedly overwrite the table for k=0, k=1, ... k=n-1.
for (let k=0; k<n; k++) {
  for (let i=0; i<n; i++) {
    for (let j=0; j<n; j++) {
      table[i][j] = Math.min(table[i][j], table[i][k] + table[k][j]);
    }
  }
}

console.log(`Shortest path length from v_1 to v_2 = ${table[1][2]}`);
```

Application - Transitive closure

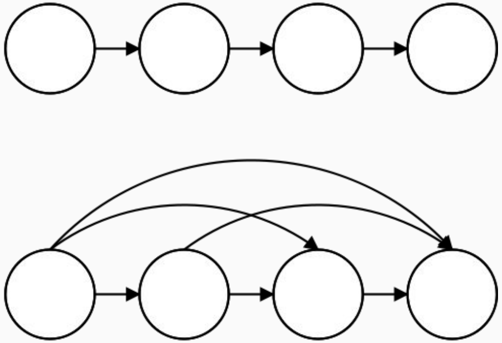
If in G there is **a path** from V_i to V_j…

…then in G^+ there is **an edge** from V_i to V_j.

G^+ is called G's **transitive closure**.

G=

G^+=



Problem:
Construct the transitive closure of a given graph.

Application - Transitive closure

Warshall's algorithm approach:

> Let $T_{ij}^{(k)}$ be true iff there's a path from V_i to V_j using only $\{V_1, V_2, \dots V_k\}$ as "internal vertices."

Wait a second, this is the same as Floyd's algorithm except we're only keeping track of booleans instead of tracking edge weights. 🤔 Let's try the same recurrence:

$$
\begin{aligned}
T_{ij}^{(0)} &= \text{True} && \text{if } (i, j) \ni E \\
T_{ij}^{(0)} &= \text{False} && \text{if } (i, j) \not\ni E \\
T_{ij}^{(k)} &= T_{ij}^{(k-1)} \lor (T_{ik}^{(k-1)} \land T_{kj}^{(k-1)}) && \text{if } k>0
\end{aligned}
$$

"Have we already found a path from V_i to V_j without using V_k,
or is there a path from V_i to V_k and a path from V_k to V_j?"

Application - Transitive closure

```
for i from 1 to |V|:
    for j from 1 to |V|:
        table[i][j][0] = True if (i,j)∋E else False

for k from 1 to |V|:
    for i from 1 to |V|:
        for j from 1 to |V|:
            table[i][j][k] = table[i][j][k-1] ||
                    (table[i][k][k-1] && table[k][j][k-1])
```

Application - Transitive closure

Implementation optimizations:

- Drop the k superscripts, just like before. (saves a factor of |V| memory)
- Get clever with writing the loops!

Remember the recursive case of the recurrence:

$$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee (T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)})$$

Notice that if $T_{ik}^{(k-1)}$ is false then logically $T_{ij}^{(k)} = T_{ij}^{(k-1)}$, i.e. **nothing can change** by introducing V_k. If we check $T_{ik}^{(k-1)}$ and it's false then we don't even need to run the inner loop over j; we just keep the same values from $T_{ij}^{(k-1)}$ in place.

But if we check $T_{ik}^{(k-1)}$ and it's true, the recurrence becomes:

$$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{kj}^{(k-1)}$$

Application - Transitive closure

```
for i from 1 to |V|:
    for j from 1 to |V|:
        table[i][j] = True if (i,j)∋E else False

for k from 1 to |V|:
    for i from 1 to |V|:
        if table[i][k] is True:
            for j from 1 to |V|:
                table[i][j] = table[i][j] || table[k][j]
```